The University of Manchester
Department of Computer Science
Thrid Year Project Report 2021

Binary-Compatibility with Linux Application for a Unikernel Written in Rust

Author: Laurent Pool

Supervisor: Dr. Pierre Olivier

Abstract

Binary-Compatibility with Linux Application for a Unikernel Written in Rust

Author: Laurent Pool
Supervisor: Dr. Pierre Olivier

This project was shared with Christopher Densham

The aim of this project is to adapt a unikernel written in Rust, RustyHermit, to be binary compatible with Linux applications. Unikernels are specialised lightweight virtual machines running a single application. They are lightweight as they only provide the bare minimum operating system functionality and resources required to run the application. It is for this reason, as well as their security benefits, that they are good contenders for use in cloud and edge computing. However, their widespread use has been limited due the effort required to port existing applications in order to run them as unikernels. A solution to this previously explored is a new model for unikernels, binary compatible unikernels, which significantly reduces the porting effort for the application.
This project explores combining the binary compatibility model, with another model of unikernels which boasts memory safety through the use of memory safe programming languages, in an effort to combine the best of these two approaches. At the end of the project we show that binary-compatibility can be achieved on a memory safe unikernel, RustyHermit, by running a "Hello World!" application and some more assembly applications, all compiled for Linux.

Acknowledgements

I would like to thank Dr. Pierre Olivier for his continuous support and guidance throughout the course of this project, Stefan Lankes for providing valuable insight into RustyHerrmit and more, and also Christopher Densham who played a large role in the development of the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

INTRODUCTION Unikernels are lightweight, fast in terms of execution and deployment time, secure internally and between other unikernels. All these benefits tick most of the boxes of many virtualisation application domains: namely HPC [20], cloud and edge deployed resources [1, 18, 3] and IoT [21]. They are also a unique way of exploring operating systems on a smaller scale which I have found out over the development of this project. However, they have so far mostly remained as niched projects and not strong contenders when it comes to choosing between different virtual machine implementations in these domains [3, 16]. One of the mains reasons for this is due to the large amount of effort required or sometimes impossibility in porting an existing application to use this approach.

There have been two main design approaches to unikernels in the past [8]. Legacy unikernels, aimed at supporting legacy applications by emulating the interfaces of other OS's and usually written in C and C++ [20, 17, 2, 22]. These languages are however not memory safe, allowing unchecked memory accesses causing common errors such as null pointer dereferencing and different types of memory leaks [24, 12]. Memory safety aside, one of these legacy unikernels, namely Hermitux [16], takes a unique approach in reducing the porting effort, and even at times eliminating it, by being binary compatible, being able to run binaries natively compiled for Linux. It does all of this whilst maintaining the unikernel principles.

The other design approach to unikernels is a clean slate approach [8, 10, 19], and contrary to legacy unikernels does not emulate any OS interface but instead emulates the necessary application interface providing support applications written in the same high level language of its library operating system kernel. These high level languages provide the benefits of memory safety [10, 2]. Unfortunately, these unikernels still require a large amount of effort in porting the application to use these interfaces.

Both of these design approaches to unikernels, clean slate and more specifically binary compatible unikernels within legacy ones, both have their benefits and downsides and interestingly, the benefits of each one seems to overcome the weakness of the other. And so a combination of the two approaches leave us with a unikernel which is binary compatible, overcoming the porting effort, and provides memory safety which is lacking in legacy unikernels. This is an attractive solution and one that we have developed using RustyHermit as a base [19].

RustyHermit would be considered as a clean slate unikernel, it is written in the Rust programming language, which is memory safe. The way in which Rust provides its memory safety is through a unique memory management method it calls "ownership" and this allows

Rust code to run faster as compared to other high-level programming languages [13, 9], which is attractive both when it comes to kernel development and in the domain of unikernels.

Binary compatibility is achieved by developing a binary loader to load the kernel and the application binary into the virtual memory address space. In addition to this, a system call handler and a set of 19 system calls were developed to support the application at runtime whilst running as a unikernel. The unikernel we end up with is binary compatibility, being able to run assembly application binaries compiled for Linux with no porting required. We demonstrate its results that the project is able to succeed in running a set of assembly applications whilst maintaining unikernel principles. However, due to issues in the development of the project relating, we are not able to support C applications.

This report, describes the design of this new unikernel hybrid, describes the development process of implementing this unikernel. We also evaluate if this unikernel we end up with actually solves the two issues of porting the application and memory safety and suggest further modifications to it for the future.

Chapter 2

Background & Related Works

2.1 Unikernels

Unikernels are customised, single address space virtual machine images constructed with an application and a library operating system providing the bare minimum kernel-functionality in order for the application to run as required. They can be executed as virtualized guests on top of a hypervisor. An application can be run as a unikernel by compiling it statically against the kernel of the library operating system.

The minimalist approach of unikernels makes their memory footprint small since it leaves out all the unnecessary and unused functionality which are required for a general-purpose operating systems to perform and run a wide variety of functions and applications. This is not the case of unikernels which aim to run a single application. The resulting small code size of the unikernel which provides these few necessary functionality has the advantage of creating a reduced attack surface making it more secure.

Since unikernels only run one application at a time there is only a single address space which allows the kernel and application to share. This has the benefit of removing the need for memory protection between applications since there is only one, which allows the application to run at the same highest level privilege as the kernel code, removing the need for costly switching between privilege levels when performing system calls.

This leads us to another advantage of unikernel, which is its speed. System calls can be performed as normal function calls since we do not have to change between privilege levels as mentioned before, and because there is no need to initialise virtual devices and services not needed by the application this makes their boot times very fast.

Another advantage which we briefly mentioned before is the security that it provides. By only having the functionality required by the application running as a unikernel there are simply less things attackers can exploit, for example there is no Shell which are commonly used by attackers to deliver malicious payloads.

Going back to its security advantages, when we have several unikernels running different applications they are inherently strongly isolated from each other due to their separate virtual address spaces managed by the hypervisor and so cannot access each other's resources.

All these benefits make unikernels attractive in the space of cloud and edge computing [1, 18, 3], high performance computing (HPC) [20], in the Internet of Things (IoT) domain [21].

2.1.1 Two models of unikernels

Unikernels can follow either one of two models [8, 10].

Clean-slate Unikernel

A clean slate approach, which does not try to emulate the interfaces of other operating systems running legacy applications but instead they are written in a single high-level language and only provide the necessary interfaces for applications in that language. This has the advantages of a high-level programming languages which has a strong memory model and typed system and able to avoid typical issues such as dangling pointers which occurs in operating systems written in C [9, 19, 10].

Legacy Unikernel

The second approach or model of unikernels are legacy unikernels. These provide support to subsets of POSIX compliant applications which would allow very little to no change to legacy applications or software for them to be run as unikernels. An advantage of such a unikernel for example one that supports Linux applications is that it is able to take advantage of battle-tested Linux software, from a highly active community which maintains Linux and fixes bugs [20, 17, 2, 22].

2.1.2 Problem of porting application

Despite all the benefits of unikernels mentioned before, they have not yet achieved mainstream adoption by the mainstream industry [3, 16]. One of the main reasons for this is that it is quite difficult to port an existing application to use the library operating system services required to be run as a unikernel. This requires extensive knowledge of the library operating system and also the application itself and is a tall ask for either of the library operating system developer or the application developer [10, 14, 3, 17]. In some circumstances it may even be impossible to port the application. For proprietary applications, only the binary executable of it is available and not the source code [16]. For large complex applications, lack of documentation and even complex build infrastructures makes it close to impossible to port them.

2.1.3 Binary Compatibility

A possible solution to reduce the effort in porting the application is binary compatibility. This solution was proposed and developed in a previous unikernel project Hermitux [16], written in C. It provides binary compatibility to Linux binaries which significantly reduces and sometimes eliminates the porting cost.

It is achieved through following the conventions of Linux's ABI [11]. The Linux ABI describes the binary interface between the compiled application and the OS. Hermitux uses these conventions to emulate how Linux interfaces with the application binary without having to recompile it. These conventions can be split into two groups and how they are use to achieve binary compatibility is described below:

Load-time Binary Compatibility

These conventions are followed when the kernel and application is loaded into the virtual memory address space by the hypervisor. The kernel is loaded at a location which will not overlap with the subsequently loaded binary application which is in ELF format. Only specific loadable segments of the binary are loaded into memory, and these segments and their final location is specified in the ELF headers of the binary. The next step is setting up the stack and the heap and is done when the kernel initialises, this is required by the C runtime to initialise the application before it is executed alongside the kernel, on top of the hypervisor.

Runtime Binary Compatibility

These conventions support the application at runtime when it invokes system calls. System calls are high privilege functions which the application asks the kernel OS to perform and in this case it will be the kernel of the library OS. To catch these system calls from the application a system call handler is installed during the initialisation of the kernel, when a system call is invoked the system call handler redirects the system call to those implemented within the kernel of the library OS.

These conventions achieve binary compatibility and Hermitux shows that it is feasible to do this whilst still maintaining the unikernel principles.

2.1.4 Problem of Memory Unsafety

Although Hermitux boasts binary compatibility which significantly reduces the porting cost for the application, it is still a legacy unikernel written in a memory unsafe language, C. Because of this it is susceptible to many bugs which comes with unchecked memory accesses, such as, access error, uninitialised variables and memory leaks [24]. Clean-slate unikernels are usually written in high-level programming languages which have many memory safety checks [10], but like we mentioned before they require large amounts of effort to port the application.

2.2 Rust & RustyHermit

2.2.1 Rust

”Rust is language is a multiple-paradigm programming language designed for performance and safety…” [28].
It was originally designed by Graydon Hoare. The Rust programming language uses a unique method of ensuring memory safety which it calls “ownership” based on the borrowing of pointer which allows memory to be managed by a set of rules checked at compile time removing the need of a garbage collector which slows down other memory safe programming languages such java [13]. This memory management approach contributes to the performance of Rust code which enables it to compete against other languages like C and C++ which are known for their performance and commonly used for OS kernel development [20, 12].

2.2.2 RustyHermit

RustyHermit is a unikernel project completely written in the Rust programming language and runs applications written in the same language as unikernels [19]. It is therefore a clean slate unikernel. Instead of using the stable version of Rust, RustyHermit uses the unstable version Nightly Rust. This provides extensions to support low-level programming which is inherent when it comes to OS development, low-level programming is used to handle interrupts, edit the stack, manipulate registers and accessing explicit memory addresses, all these are not supported by the stable version of Rust.

Chapter 3

Motivation

3.1 Solution

A unikernel written in a high-level programming language providing memory safety being binary compatible brings together the best of a clean-slate unikernel and a binary compatible unikernel, whilst solving the main issues the two faced individually. This project explores this type of unikernel and explores whether this is feasible.

3.2 Why RustyHermit?

The RustyHermit unikernel is an attractive base to implement this unikernel. The main reason it was chosen over other clean-slate unikernels is that it is written in Rust. A memory safe programming is what is needed for this project, that being said, performance of a language is one of the main reasons why they are chosen for OS development, and it is the performance of Rust which makes it stand out amongst other memory safe programming languages. RustyHermit, being written in Rust, makes it the clear winner.

Chapter 4

Aims & Objectives

This Chapter explicitly states our aims and objectives for this project.

4.1 Aims

The aim of this project is to demonstrate the feasibility of the unikernel we mention in the motivation section. This is done by extending the RustyHermit unikernel to be binary compatible with Linux binaries whilst taking advantage of the memory safety features and performance benefits of the language it is written in, Rust.

4.2 Objectives

4.2.1 Objectives for Binary Compatibility

To achieve binary compatibility the main objectives includes:

- Developing a loader within RustyHermit to load and execute the an application according to Linux's Application Binary Interface
- Developing a system call handler within RustyHermit to catch system calls made by the running application at runtime
- Implementing a set of system calls within RustyHermit to support a targeted set of simple applications
- Validating the system calls work as expected by explicitly testing specific use cases

4.2.2 Applications we want to support

However, achieving full binary compatibility is a very large task which requires almost the full re-implementation of Linux. This would of course require more time than that provided for the project and is ultimately not our goal, and so described below is the order and extent to which we wanted to provide binary compatibility through providing support to be able to run the for the following applications:

- A statically compiled Hello world application written in assembly

- A statically compiled Hello world application written in C
- A small subset of statically compiled benchmark applications written in C

The benchmarks mentioned above are compute/memory intensive HPC benchmarks with few demands in terms of OS functionality. These play a role in evaluating our final product, not only helping us in determining if this solution is possible, but also if it's worth it.

This project was available to two students, Christopher Densham and myself, and as such the work was split up such that Christopher worked on the load-time binary compatible conventions which involves developing the loader. The runtime binary compatibility convention was done by myself, and involves development of the system call handler and the system calls.

Chapter 5

Planning

This section describes the plan developed to complete this project.

5.1 How the work was split up

As mentioned before, the project was split up between Christopher and myself. The job of supporting the load-time binary compatible conventions was given to Christopher and supporting the runtime binary compatible conventions was given to me. We can see from the chart in figure (5.1) that these two tasks are almost completely independent of each other enabling us to work independently for a large duration of the project.

This report will only provide brief descriptions of Christopher's parts of the project and I would refer the reader to his report for a more comprehensive description of what his part entails and how it was implemented.

It remains to be said that the chart in figure (5.1) was made at the very beginning of the project, and so does not represent well the amount of work required and time spent on each part.

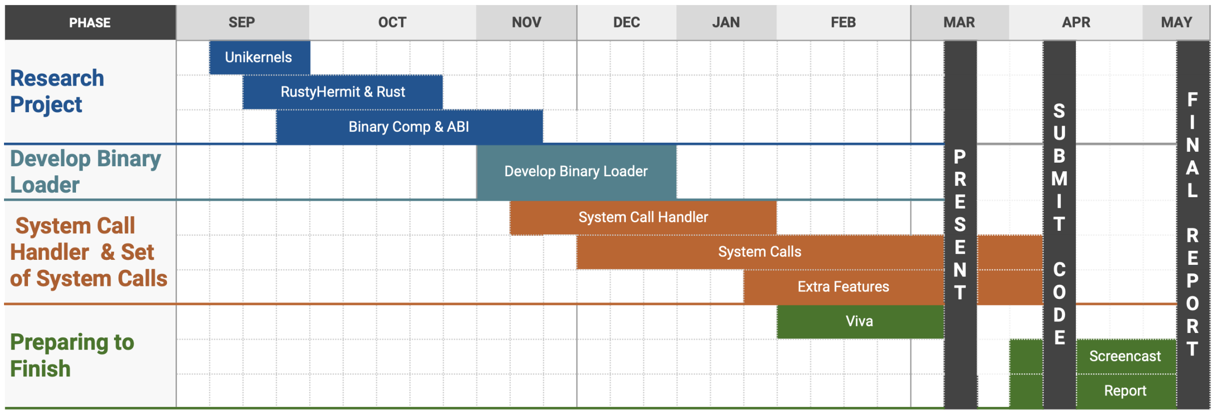


Figure 5.1: Initial Gantt chart which was used as a very rough guide before the submission dates of the project changed.

5.2 System call Handling

The first task for myself, was to enable system call support at boot time by developing and installing a system call handler. After developing the system call handler, the order and specific system calls implemented were guided by the order of the applications we wanted to support, mentioned in the Aims and Objectives section. The order of these applications are such that the system calls they use are generally increasing in complexity to implement.

Chapter 6

Design

This section describes the design of the whole project as a whole, then describes the design of its different major elements.

6.1 System Overview

Currently, the project is only able to run binaries of x86_64 assembly code which is not linked to a C library and as such is inherently static. More on this in the Implementation section.

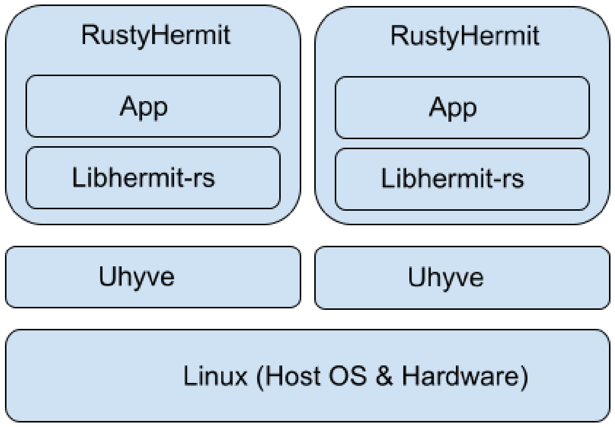


Figure 6.1: Design of the overall system of our RustyHermit.

As shown in the figure (6.1), RustyHermit runs on top of the minimal hypervisor called Uhyve which in turn runs on top of our host OS, Linux. Before running the application as a unikernel, the hypervisor must request memory for the unikernel. The virtual address space of the kernel, Libhermit-rs, and application will be mapped to this memory. The kernel is then loaded into the address space by the developed binary loader. The application binary is then loaded into this address space and is done by the binary loader. At this point we begin to execute the kernel code which initialises all the facilities required to run the process, this includes registering the system call handler developed. After the completion of the initialisation of the kernel we jump to the beginning of the application and it begins its execution as a unikernel while the kernel emulates Linux services by catching system calls and running our own implementations of them to achieve the same behaviour from the application whilst providing unikernel benefits.

6.2 Binary Loader

After the hypervisor has obtained memory from the host which will be the address space, the kernel is then loaded into it such that it will not overlap with the application, this is shown in figure (6.2). It is then followed by the binary application which is in ELF format [5]. Not all of the application is loaded into the address space, only the loadable segments of this binary file which is specified by the ELF metadata and is loaded at another address also specified by the Linux ABI [4, 11]. This next step involves creating the stack and initialising it with values that are required by the C runtime to initialise the virtual address space for the application, also shown in figure (6.2). Finally, we jump to the entry point of the application so it can now be executed as a unikernel. The binary loader was developed by Christopher.

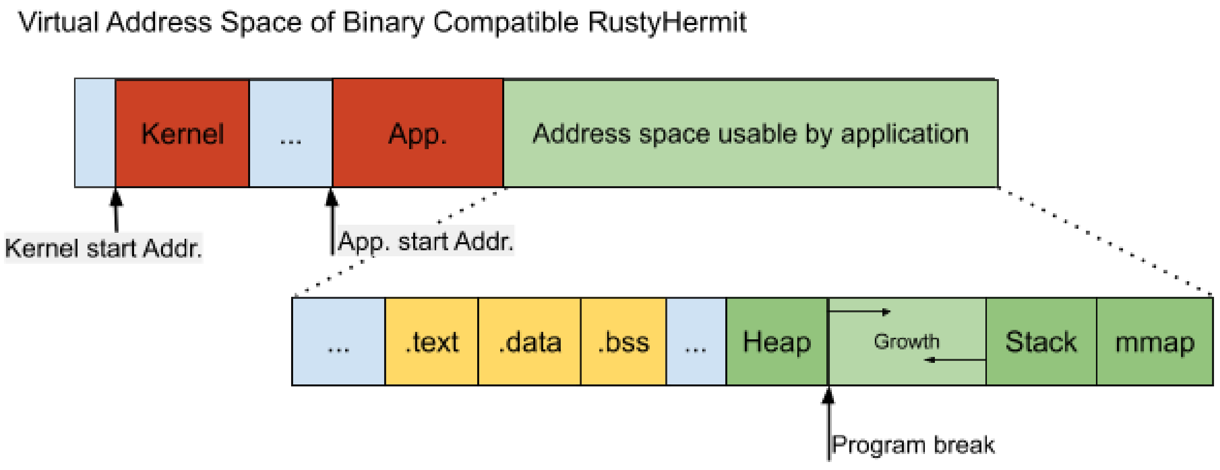


Figure 6.2: The virtual address space containing both the kernel and the application.

6.3 System Call Handler and System Calls

During the initialisation of the kernel, system calls are enabled and the our newly developed system call handler is registered. In a traditional unikernel, system calls are performed as normal function calls while maintaining the highest privilege as the kernel code. However, to be binary compatible, our implementation is different. When system calls are invoked by the application they are trapped by the system call handler, shown by [1] in figure (6.3). The system call handler saves the state of the registers then identifies the system call and redirects to the corresponding system calls. After their execution, the state of the registers are restored and execution returns to the application code where the system call was made shown by [2] in figure (6.3). The privilege level of the execution of the application and kernel remains the same during this new method of invoking system calls.

To be able to support the applications we want to run as unikernels, we need to provide implementations for the system calls they invoke. Some of these system calls were already partially implemented previously and these, including the new ones we implemented, would have to be called from the system call handler. Some of these system calls also need to interact with the host OS files and thus have to be forwarded from the kernel to the hypervisor and

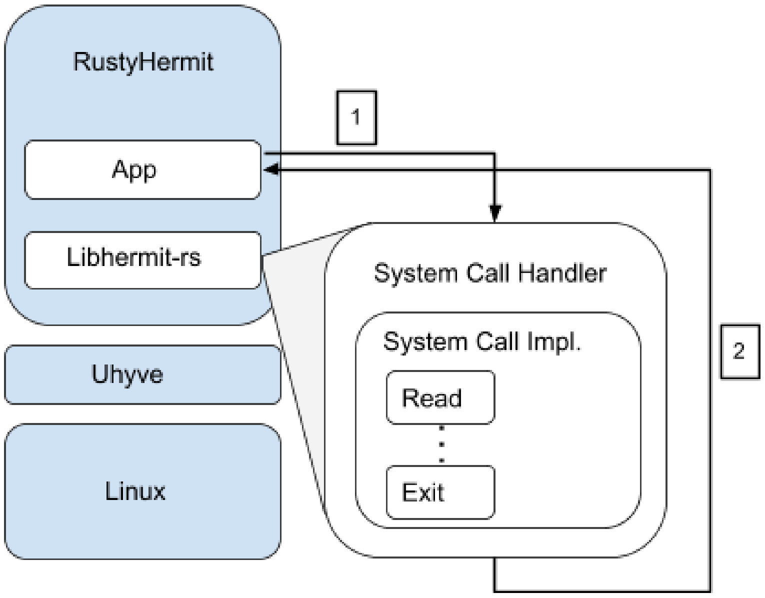


Figure 6.3: Execution of then system calls are invoked.

finally to be executed by the host's original corresponding system call. The system call handler and system calls were developed by me.

Chapter 7

Implementation (System Call Handling)

This part of the report goes over the development and implementation of the system call handler and specific system calls, and will not cover the development of the binary loader by Christopher, I would refer the reader to his report for the implementation details.

7.1 Enabling and Installing System Call Handler

Linux applications uses the "syscall" x86_64 instruction for system calls, and initially the Libhermit-rs, the kernel of the LibOS, does enable system calls during the initialisation phase of the kernel and the next step would be to install the system handler according to [7]. This is done by setting the model specific register, `MSR_LSTAR`, to the address of the entry point of the system call handler. When a syscall instruction is invoked from the application code, the next address loaded into the instruction pointer register comes from this `MSR_LSTAR`. By placing the address of the entry point of the handler within this register we are turning the invocation of the syscall instruction to a call to our system call handler enabling us to catch every system call made by the application at runtime. Two other registers need to be changed, namely `MSR_STAR` and `MSR_SYSCALL_MASK`.

7.2 System call Handler

Before a syscall instruction is invoked from the user code, the registers have some state and may hold arguments for the system call or values which may be required after the system call is executed on behalf of the kernel when returning to the user code. In addition to this, when a syscall instruction is invoked, the execution flags and the user code address to return to after, are also stored in registers. Because of these, we need to make sure that we preserve the state of the registers during the execution of the system calls.

Preserving the state of registers constitutes the first part of the system call handler, here called isyscall written entirely in assembly, the first half of is shown in figure (7.1). This function is also the entry point to the system call handler and thus the first thing executed when a syscall is invoked. The register values are pushed onto the stack and will remain there during the execution of the system call, by doing this we create a copy of their values which is stored on the stack. Now we do not have to worry about losing the register values when we execute the system call since we can later restore them from the copies placed on the stack. In addition to this, interrupts are disabled before we push the registers and enabled again before we jump to the second part of the system call handler. This is to prevent other possible interrupts from preempting the register saving process which could potentially cause register values to be changed and lost. The register values are pushed onto the stack in such a way that we can access the initial specific register values from the second part of the system call handler.

```
 1  .global isyscall
 2  isyscall:
 3          cli
 4          sub $0x100, %rsp        // Create some room to use, to  not mess up stack
 5          push %r15
 6          push %r14
 7          push %r13
 8          push %r12
 9          push %r11               // rflags
10          push %r10
11          push %r9
12          push %r8
13          push %rdi
14          push %rsi
15          push %rbp
16          push %rbx
17          push %rdx
18          push %rcx               // Return address
19          push %rax               // First variable to struct
20          mov %rsp, %rdi          // Address of struct on stack
21          sti
22
23          call syscall_handler
24
```

Figure 7.1: Entry point of the system call handler, showing register values being pushed onto the stack in order to save their initial values. Second part of system call handler is called on line 23.

After saving the state of the registers we jump to the second part of the system call handler shown in the (7.1) on line 23. This part of the handler, partly shown in figure (7.2), is a large case statement on the RAX register which is amongst the registers we pushed onto the stack earlier. Linux ABI identifies the RAX register as the register used by the application code to specify the system call which it wants executed. The value representing this is loaded into this register before the syscall command is invoked along with arguments which may be required by the specific system calls and these are loaded the exact order of the following registers; RDI, RSI, RDX, R10, R8 and R9.

The initial value of the RAX register stored on the stack value is accessed on line 33 of figure (7.2) through the "State" struct shown in the signature of the function on line 31 of the same figure. Subsequently, the system call number is matched to its corresponding system call implementation within the match statement. We then call the function implementing this system call providing its corresponding arguments which are also accessed from the "State" struct.

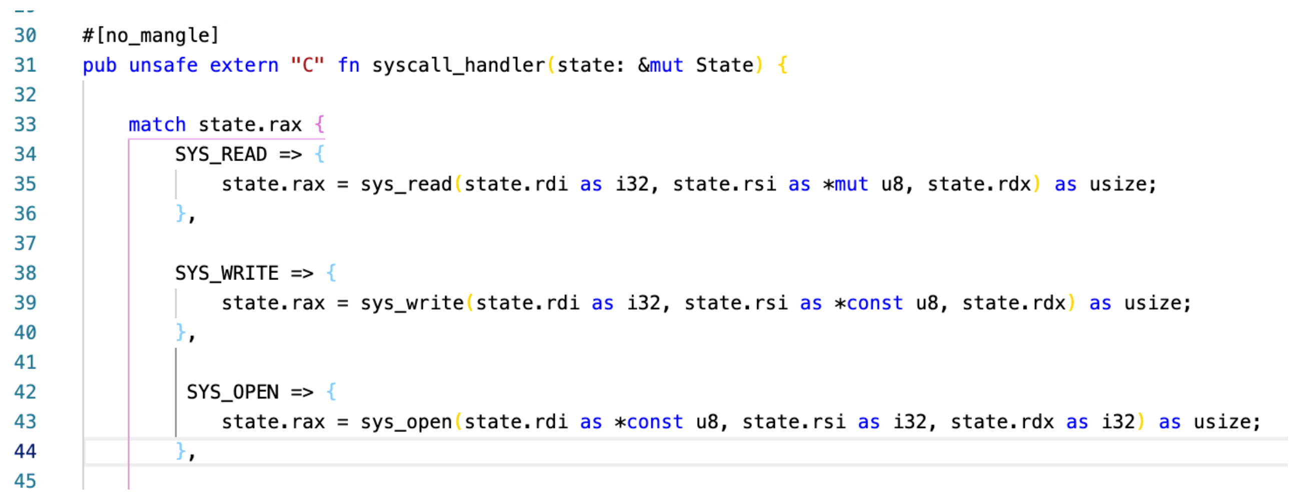

```
30  #[no_mangle]
31  pub unsafe extern "C" fn syscall_handler(state: &mut State) {
32
33      match state.rax {
34          SYS_READ => {
35              state.rax = sys_read(state.rdi as i32, state.rsi as *mut u8, state.rdx) as usize;
36          },
37
38          SYS_WRITE => {
39              state.rax = sys_write(state.rdi as i32, state.rsi as *const u8, state.rdx) as usize;
40          },
41
42           SYS_OPEN => {
43              state.rax = sys_open(state.rdi as *const u8, state.rsi as i32, state.rdx as i32) as usize;
44          },
45
```

Figure 7.2: Second part of system call handler which calls the specific system call implementation with the correct set of arguments.

When system calls are executed, they usually return a value, this value may represent the status of the execution of the system call, and it must be returned to the application within the `RAX` register. One instance, on the figure (7.2), where this is done is on line 35, where the rax variable of the "State" struct is changed to the value returned by the function representing the system call implementation. Doing this will change the value which is stored on the stack, however the `RAX` register is one of the registers whose values are not preserved during the execution of a system call, since it will contain the return value of the system call when we return to the application code.

After this we return to the second part of isyscall, in figure (7.3) , where the state of the registers are restored to their initial value, except for the `RAX` register, and jump back to the user code to continue the execution of the application.

7.3 System call development

Like we mentioned in the planning section, the system calls we implemented and their order was determined by the applications we wanted to support. The first application we chose was unsurprisingly a "Hello World" application written in assembly. This contains the write and exit system calls, which brings us to the set of system calls which was implemented. The table (7.1) shows all the system calls we supported or partially supported which would allow us to run our chosen applications.

7.3.1 Supporting Hello World in Assembly

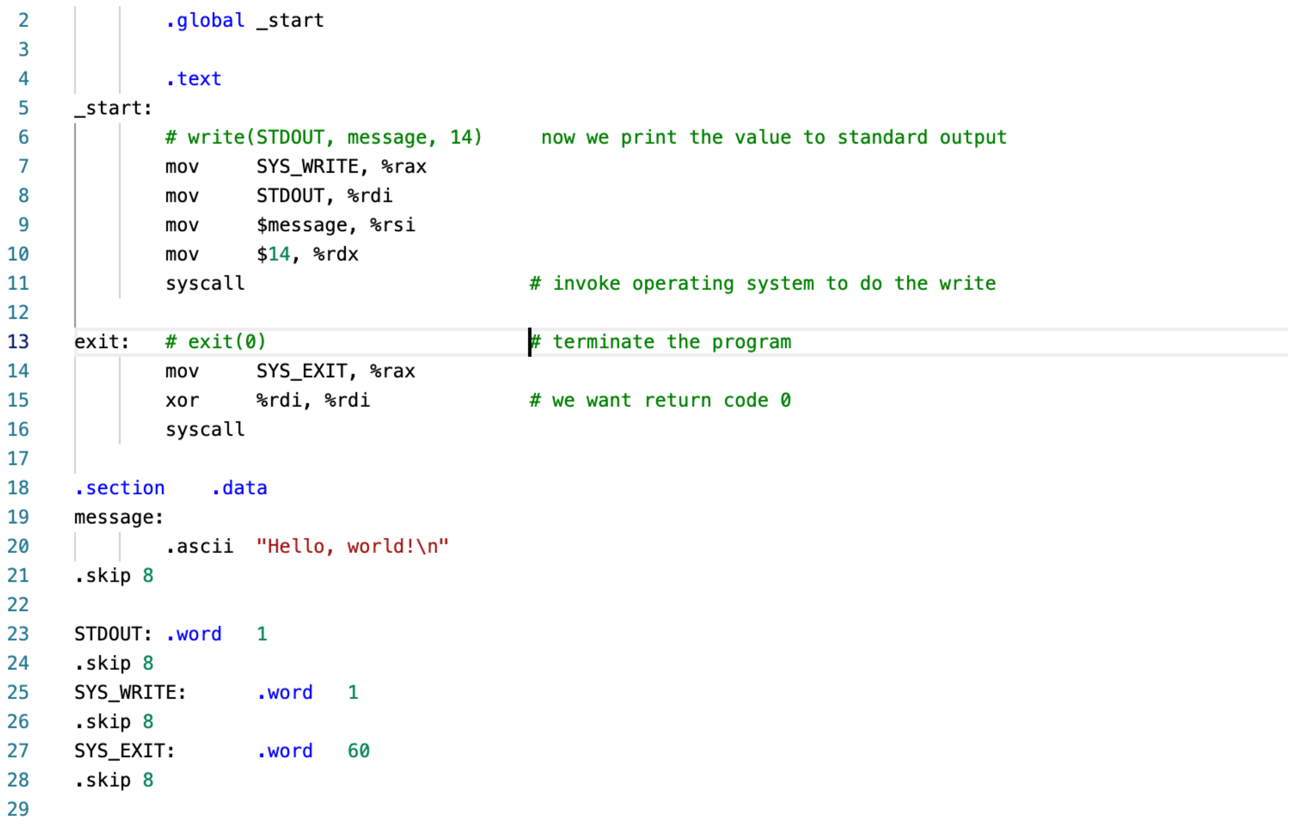
The "Hello World" application in assembly calls the write and exit system calls, shown on figure (7.4) on lines 11 and 16. Write can be considered a file management system call as it can be used to write to files, including the standard output. Many applications may interact with each other or the host OS through files, and so file management system calls were the first set of system calls implemented to support or at least partially support a large set of applications, including our "Hello world" application.

```

call syscall_handler

cli
pop %rax
pop %rcx                // Address in application code
pop %rdx
pop %rbx
pop %rbp
pop %rsi
pop %rdi
pop %r8
pop %r9
pop %r10
pop %r11
pop %r12
pop %r13
pop %r14
pop %r15
push %r11
popfq                   // Restore rflags
add $0x100, %rsp        // Discard used space on stack
sti

jmp *%rcx               // Jump back to application code
```

Figure 7.3: Second half of isyscall from figure (7.1), showing saved register values being restored into their respective register then jumping back to the application code which invoked the system call.

Implemented System Calls	
File Management	Memory Management
0. Read**	9. Mmap
1. Write**	10. Mprotect*
2. Open**	11. Munmap
3. Close**	12. Brk
4. Stat**	Miscellaneous
5. Fstat**	16. Ioctl*
8. Lseek**	60. Exit
19. Readv	63. Uname*
20. Writev	96. Gettimeofday
89. Readlink**	158. Archprctl

Table 7.1: System calls supported. * represents system calls which were faked, ** represents system calls which were forwarded and executed by the host OS.



```
        .global _start

        .text
_start:
        # write(STDOUT, message, 14)     now we print the value to standard output
        mov     SYS_WRITE, %rax
        mov     STDOUT, %rdi
        mov     $message, %rsi
        mov     $14, %rdx
        syscall                         # invoke operating system to do the write

exit:   # exit(0)                       # terminate the program
        mov     SYS_EXIT, %rax
        xor     %rdi, %rdi              # we want return code 0
        syscall

.section    .data
message:
        .ascii  "Hello, world!\n"
.skip 8

STDOUT: .word   1
.skip 8
SYS_WRITE:      .word   1
.skip 8
SYS_EXIT:       .word   60
.skip 8

```

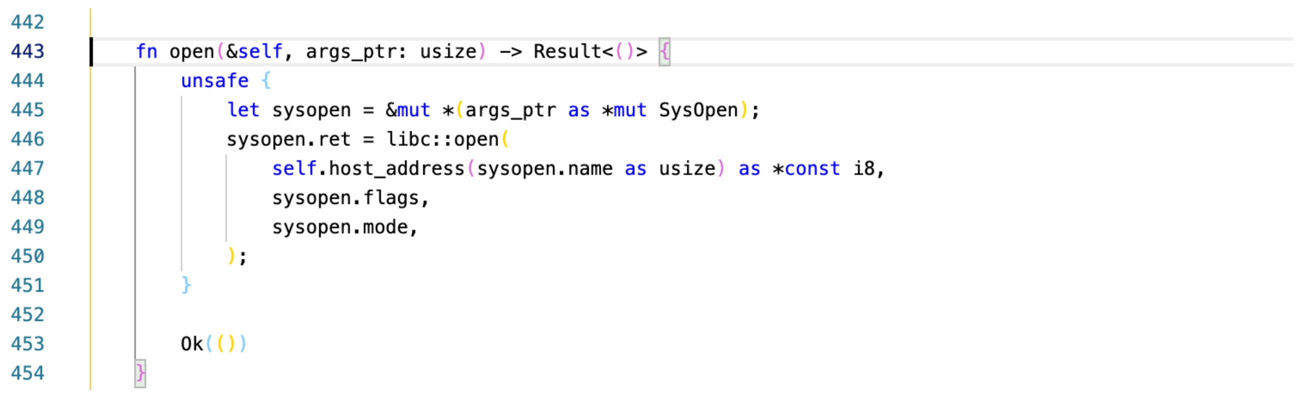
Figure 7.4: Hello_world application in assembly.

Initially the kernel of RustyHermit, Libhermit-rs, had already implemented some system calls and these included read, write, open, close, exit and other ones including more file management ones. However, in the original RustyHermit, these system calls are called more directly.

Like we mentioned before, file management system calls interacts with the filesystem of the host OS. Since system calls on the unikernel are executed from the kernel of the guest machine running on top the hypervisor these files are out of reach and cannot be accessed. The way in which the original RustyHermit deals with this is to forward these system calls to the hypervisor from the kernel, and then executed using a library called Libc, which under the hood calls the corresponding system call within the kernel of the host OS. An example of this is shown in figure (7.5), which shows the code of the hypervisor, Uhyve, redirecting the open system call to the host using Libc on line 446.

When it comes to making RustyHermit binary compatible, system calls have to be called from the handler we developed before. Instead of reimplementing new system calls, we chose to reuse those file management ones we mentioned before by calling them from the handler with the proper arguments that had been placed on the stack previously and pass the return value from the system call implementation back.

So now we have implemented several namely read, write, open, close, lseek, and exit. Write and exit required by our "Hello world" application in pure assembly.

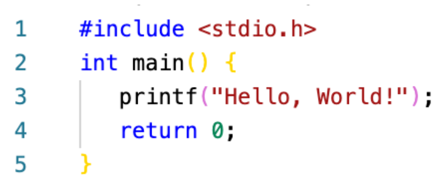


```
fn open(&self, args_ptr: usize) -> Result<()> {
    unsafe {
        let sysopen = &mut *(args_ptr as *mut SysOpen);
        sysopen.ret = libc::open(
            self.host_address(sysopen.name as usize) as *const i8,
            sysopen.flags,
            sysopen.mode,
        );
    }

    Ok(())
}
```

Figure 7.5: Code in hypervisor, Uhyve, showing open system call being forwarded to host OS's implementation, after previously being forwarded to the hypervisor itself from kernel of RustyHermit.

7.3.2 Supporting Hello World in C

After supporting the application in assembly, the next step would be to support the same application but in C. In figure (7.6) we show the code for a "Hello world" application written in C. It is evident that the code does not explicitly invoke system calls with the syscall instruction. This is because C is a higher level programming language and provides a higher level of abstraction compared to assembly. However, we can determine the system calls made by this application by using the tool "strace". Following from figure (7.7), we first compile the application statically, since we only want to support statically linked binaries. When we run this binary with the tool it outputs the system calls made by the program. Note that the output of the tool is mixed with the output of the application. Highlighted in blue is the output of the application.

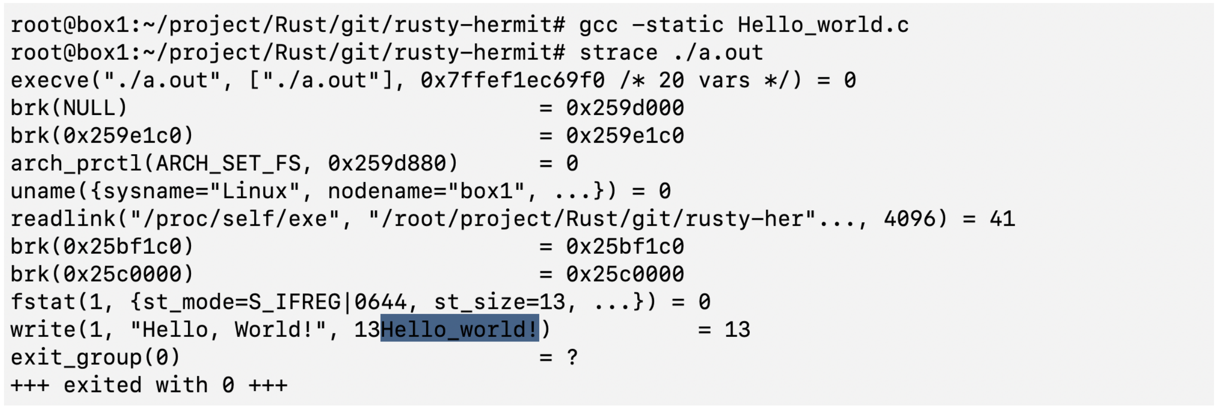


```
#include <stdio.h>
int main() {
    printf("Hello, World!");
    return 0;
}
```

Figure 7.6: Hello_world application in C.

We can see that the "Hello world" application in C makes more system calls than that in assembly, in addition to write and exit which is common to both, it also calls brk, arch_prctl, uname, readlink and fstat. This is due to the C runtime environment which is compiled together with the application code. The C runtime environment is executed before the application code is run and configures the applications memory. The C runtime performs several tasks such as setting up the on the user side and initialising the heap for the application. It is when the C runtime environment code is performing these operations that the system calls are invoked.

We are ignoring execve since this system call is not called from the application code, but by the shell in order to execute the program.



```
root@box1:~/project/Rust/git/rusty-hermit# gcc -static Hello_world.c
root@box1:~/project/Rust/git/rusty-hermit# strace ./a.out
execve("./a.out", ["./a.out"], 0x7ffef1ec69f0 /* 20 vars */) = 0
brk(NULL)                               = 0x259d000
brk(0x259e1c0)                          = 0x259e1c0
arch_prctl(ARCH_SET_FS, 0x259d880)      = 0
uname({sysname="Linux", nodename="box1", ...}) = 0
readlink("/proc/self/exe", "/root/project/Rust/git/rusty-her"..., 4096) = 41
brk(0x25bf1c0)                          = 0x25bf1c0
brk(0x25c0000)                          = 0x25c0000
fstat(1, {st_mode=S_IFREG|0644, st_size=13, ...}) = 0
write(1, "Hello, World!", 13Hello_world!)            = 13
exit_group(0)                           = ?
+++ exited with 0 +++
```

Figure 7.7: System calls made by Hello_world application in C shown by strace tool

Readlink

Readlink was one of the system calls that had already been implemented previously and like before, we have to call its implementation from our system call handler. Readlink is used to print the value of a symbolic link in a filesystem, this makes a file management system call and its original implementation in RustyHermit is to forward it to the underlying host. However, the figure (7.7) also shows a specific use of readlink when it is called with the argument "/proc/self/exe". This is sometimes used in the place of `argv[0]` in C applications.

During a normal execution of an application, calling readlink with this argument would return the path of its executable file being executed, however, now we want to run the application as a unikernel and it is the hypervisor which is currently running on the host, shown in figure (6.1) in the Design chapter. Because of this, readlink would return the path of the executable of the hypervisor, Uhyve. This is not the behaviour the application expected and does not emulate the way Linux would interact with the application running directly on top of it. To fix this, since the system call is forwarded to the hypervisor, we explicitly catch the case where readlink was invoked with this argument and returned the absolute path of the executable file of the application instead of that of the hypervisor.

Fstat

Fstat follows the implementation of the stat system call which had previously been implemented with slight variations in their arguments. Fstat is also a file management system call and is forwarded to the underlying host.

Uname

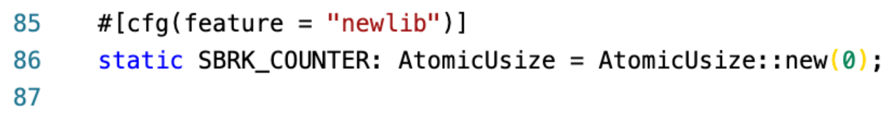
Uname is another system call and used to get information about the current kernel, these include the operating system name and its version. This system call does not provide much use to the applications we support, and in the "Hello World" application in C it is called by the C library during preparation to run the application. The decision was made to fake this system call and return hardcoded results which allowed the initialisation of the application to continue. These hard coded values were copied from Hermitux [15].

Arch_prctl

The arch_prctl system call sets the architecture specific thread state. As we can see in figure (7.7) it is called with the argument “ARCH_SET_FS” which sets the FS register, we not will delve deeper into what this is for. For the implementation of this system call we only provide support for when arch_prctl is called with arguments “ARCH_SET_FS” and “ARCH_GET_FS”. All other invocations of this system call with other arguments will emit a warning and return a value -ENOSYS which lets the application code know that this system call was not implemented. When system calls are performed there is no guarantee that it will succeed, and so applications requesting them should take this into account and so it is left to the application side to carry on afterwards. That being said, for the applications we currently want to support, we know that they will not call arch_prctl with other arguments.

Brk

Brk system call changes the heap size. It is used to allocate memory for a process, and in this case it would be used to increase the user heap, it does this by changing the program break shown in figure (6.2). The original Libhermit-rs had already implemented sbrk which is similar to brk in that they both change the program break, and so we decided to base our implementation of brk off of sbrk to change the program break which was stored in the variable `SBRK_COUNTER` shown in figure (7.8) on line 86. This brought us to our first major challenge when it comes to system calls. In figure (7.8) on line 85 and figure (7.9) line 93, we see the variable and function have a compiler directive before them which means that they would only be compiled if we provide the flag “newlib” upon compilation and without it they would not be accessible and essentially be non-existent. This directive appears throughout Libhermit-rs, and for the implementation of brk, it is sufficient to say that it also preceded by this directive shown in figure (7.10) 117, in order to access this program break.

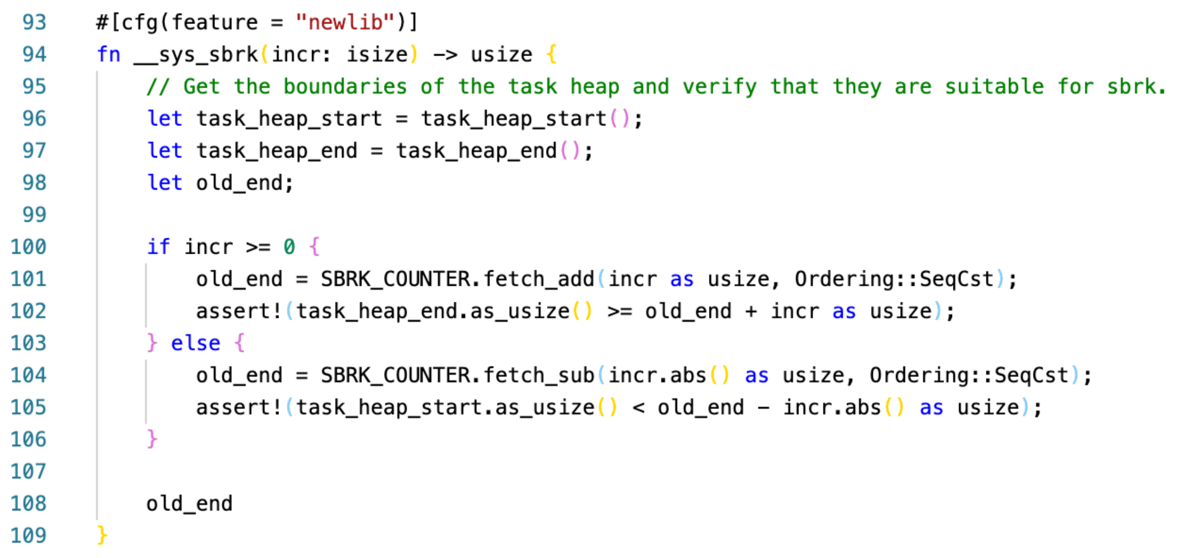


```
85    #[cfg(feature = "newlib")]
86    static SBRK_COUNTER: AtomicUsize = AtomicUsize::new(0);
87
```

Figure 7.8: Variable used to store the program break at marking the end of the heap, compiled only if newlib flag is provided.

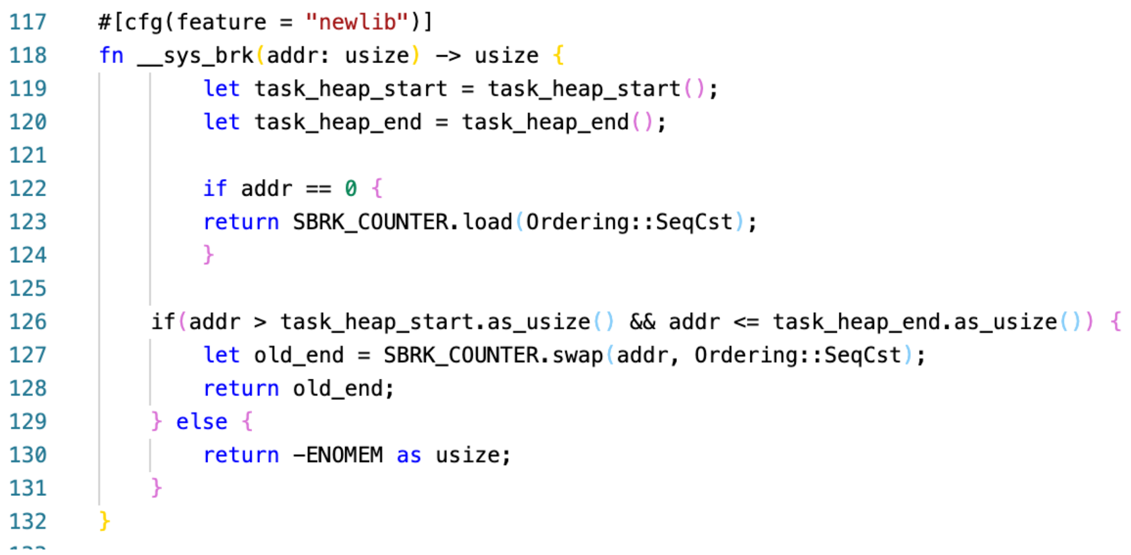
7.3.3 Supporting other applications

Other applications we wanted to support are all written in C, and they included several NPB applications [27], which are a set benchmarks and would allow us to evaluate the performance of our unikernel compared to running the applications on Linux. In addition to the benchmarks, MicroPython [25] seemed like an attractive application as it would allow us to support another programming language, Python, with minimal resources as MicroPython was made for embedded systems and this would complement the minimalist principles of unikernels. Although not all the systems calls required by these applications were supported, a few of them were and their implementation is described below.



```
#[cfg(feature = "newlib")]
fn __sys_sbrk(incr: isize) -> usize {
    // Get the boundaries of the task heap and verify that they are suitable for sbrk.
    let task_heap_start = task_heap_start();
    let task_heap_end = task_heap_end();
    let old_end;

    if incr >= 0 {
        old_end = SBRK_COUNTER.fetch_add(incr as usize, Ordering::SeqCst);
        assert!(task_heap_end.as_usize() >= old_end + incr as usize);
    } else {
        old_end = SBRK_COUNTER.fetch_sub(incr.abs() as usize, Ordering::SeqCst);
        assert!(task_heap_start.as_usize() < old_end - incr.abs() as usize);
    }

    old_end
}
```

Figure 7.9: Original implementation of sbrk system call using, compiled only if newlib flag is provided.



```
#[cfg(feature = "newlib")]
fn __sys_brk(addr: usize) -> usize {
        let task_heap_start = task_heap_start();
        let task_heap_end = task_heap_end();

        if addr == 0 {
        return SBRK_COUNTER.load(Ordering::SeqCst);
        }

    if(addr > task_heap_start.as_usize() && addr <= task_heap_end.as_usize()) {
        let old_end = SBRK_COUNTER.swap(addr, Ordering::SeqCst);
        return old_end;
    } else {
        return -ENOMEM as usize;
    }
}
```

Figure 7.10: Newly developed brk system call implementation based off of sbrk, compiled only if newlib flag is provided

Ioctl

Ioctl is used to control the input and output for devices. Without going into further detail, we chose to fake the success of the system call by hard coding and returning the values returned when executed by Linux, similarly to what we did for the uname system call.

Gettimeofday

Gettimeofday was again previously implemented by the original Libhermit-rs, and we added a call to its implementation within our system call handler.

Readv & Writev

Readv and writev are system calls commonly used by applications compiled and linked with the musl library[26]. They operate similarly to the read and write system calls except they perform the read and write using several buffers. Because readv and writev are essentially performing several read and write system calls, they have been implemented as wrappers around their corresponding, more atomic counterpart.

Mmap

The simplest use of the mmap system call is to request memory for the running application similar to brk and sbrk, however the memory is not on the heap and so does not affect the program break. It does this by requesting virtual pages for the application and it may or may not be linked to physical pages. Functions like malloc use both brk and mmap to request more application memory, however, brk is used for smaller allocations and mmap will be used for larger allocations.

For the implementation of mmap, we chose to only implement the simplest use case for now as a means to learn more about this system call and lay the groundwork for it as it would provide several benefits for the project in the future. For now this mmap requests for physical memory pages and also virtual memory pages for the application at any address and maps these to each other enforcing access writes for these pages according to the arguments passed by the user-side code. The base address of these mapped pages are then returned to the application.

Munmap

Munmap does the reverse and deallocates the pages allocated through mmap and its implementation does the same.

Mprotect

Mprotect is used to change the access writes of memory obtained through a mmap invocation, however, the success of its invocation is faked in our implementation and a warning message is outputted when it is called.

7.3.4 System call development in Rust

Unsafe code

When it comes to developing system calls in Rust, unsafe code blocks are used. These are used to gain as much control as C when it comes to dereferencing raw pointers and unchecked memory accesses [19]. The memory management method of Rust adds checks at compile time to ensure that memory is being used safely. However, due to the nature of how the arguments of system calls are passed to the kernel from the application code, which is through raw pointer in registers, this requires regular dereferencing of pointers and casting them to specific types to be able to manipulate them. The compile time checks cannot ensure that at runtime these pointers from the application code are not null references, and so they are considered unsafe and the compiler will output error messages upon compilation. The way the original RustyHermit deals with this is to use unsafe code blocks around areas of code which do perform operations which cannot be checked at compile time. Using the unsafe code blocks removes the compile time checks for these areas and allows the code to perform these operations.

The use of unsafe code blocks is minimised in the original RustyHermit as these areas represent parts of the code where no memory checks are performed which is inherently unsafe and more care should be taken by the developer when performing these operations. The new system calls supported above follows the same approach to minimise the use of unsafe code blocks.

7.4 Challenges

During the development of the project we faced many challenges, some of these significantly impeded our progress.

7.4.1 Unstable Nightly Rust

The first challenge we faced started before beginning the development of the project. When we first looked at the RustyHermit unikernel it could not compile. RustyHermit is written in Nightly Rust which is unstable, and what caused this issue was that one of the libraries, which rusts calls crates, which it depends on, had been updated. This update was such that the program could no longer compile due to code within the project now using old signatures of functions of the crate which had now been updated to use different arguments. The fix for this was to explicitly lock the crate to its previous version so that any further update to this crate would not affect our project. I do not provide the name of the crate and the functions as I did not record this at the time it happened. It was especially hard to overcome this problem, as at the time we were new to the Rust programming language, let alone Nightly Rust. Throughout the development process of the project, issues like this arose several times, which impeded our progress.

7.4.2 Newlib and LWIP

In the implementation of the brk system call we mentioned the "newlib" compiler directive. Any function or variable which was preceded by this directive would not be compiled unless the "newlib" flag was provided at compilation of the kernel of RustyHermit, Libhermit-rs. However, with the complex build infrastructure of RustyHermit, this flag was not provided during the compilation. This meant that we would not be able to support the brk system call as the dependencies within its implementation required interactions with variables and functions compiled with this flag. One example is that brk accesses the variable SBRK_COUNTER which we can see from figure (7.8) line 85 is also only compiled if we provide the specific flag.

In an attempt to solve this problem, the compiler flag was added to the build process of RustyHermit in order to compile its Libhermit-rs with this flag. This proved not to work as the build process failed, outputting several errors relating to uninitialised symbols namely "init_lwip", "lwip_read" and "lwip_write". LWIP is a minimal and lightweight TCP/IP stack implementation [23]. These errors means newlib requires support for LWIP, however this is not available in RustyHermit. This means that we are still unable to provide support for our brk system call required by the “Hello world” application in C.

Instead of removing the compiler directives off the functions and variables which could potentially enable use to support brk, we chose to use another unikernel project which supports LWIP. This is because LWIP would enable the project in the future to implement networking system calls whilst maintaining a small resource footprint since LWIP was designed for low resource domains such as embedded systems.

Moving to another unikernel project does not mean that we now have to perform all the same steps in extending RustyHermit to provide binary compatibility to another unikernel. This is because of Hermit-playground.

7.4.3 Hermit-Playground

The Hermit-playground unikernel is targeted to scalable and predictable runtime for high-performance and cloud computing [6]. This unikernel is written in Rust and runs C applications. Hermit-playground is very closely related to RustyHermit in that the kernel of both of these library OS’s is Libhermit-rs. This means that moving to Hermit-playground from RustyHermit not does require that much effort. Most of the effort is in copying our extended Libhermit-rs in RustyHermit to Hermit-playground.

Hermit-playground supports LWIP and its different build infrastructure compiles Libhermit-rs with the “newlib” flag and is what we were looking for. Using Hermit-playground would allow us to support the brk system call and would allow network system calls to be developed in the future.

7.5 Developing System calls Tests

7.5.1 In Rust

When we develop these system calls, we want their behaviour to emulate those of the Linux kernel so that the application running as the unikernel has the same behaviour as it would have running directly on a Linux host.

Whilst implementing these system calls mentioned above, binary compatibility with assembly applications had not been achieved yet, although the system call handler had already been developed and registered in the initialisation of the kernel. The only way to test these system calls for now is to develop Rust applications which invokes the syscall instruction.

To explicitly call the syscall instruction and place arguments to these register values within specific registers we need to use inline assembly. This allows us to embed assembly code within the application code which is in a higher programming language. Nightly Rust supports inline assembly and can be enabled by specifying `#![feature(asm)]` at the beginning of applications which makes use of it.

Figure (7.11) shows an example of calling a write system call within assembly. We can see from the diagram we are following the conventions of the Linux ABI in placing the arguments to the system call within specific registers. On line 22 the value representing the write system call is placed within the `RAX` register. It is also evident from the same figure (7.11), that we use an unsafe code block. This is because the Rust compiler cannot perform checks on assembly language and cannot guarantee that its use is safe.

```
    unsafe{
        asm!("syscall",
            in("rax") sys_write,
            in("rdi") fd,
            in("rsi") message_ptr,
            in("rdx") message_len,
            lateout("rax") ret_value
        );}
```

Figure 7.11: Example of how write system call is invoked using inline assembly in Rust code

Tests for some of the system calls mentioned above were developed in Rust using inline assembly. However these are not exhaustive tests, and only verifies the basic use of these system calls work as expected.

7.5.2 In C

The brk system is not available in RustyHermit, this is described under the challenges section of the this chapter. This means that we cannot perform tests on this system call. However, Hermit-playground does support this system call and the test for brk was written for Hermit-playground. The same principles for developing the tests in Rust is followed by here and figure (7.12) shows an example of how the brk system call is called in inline assembly.

```
asm volatile (
        "mov    sys_brk, %%rax\n\t"
        "mov    $0, %%rdi\n\t"
        "syscall\n\t"
        "mov    %%rax, ret_g\n\t"
        :
        :
        : "%rax", "%rdi"
);
```

Figure 7.12: Example of how write system call is invoked using inline assembly in c code

7.6 Putting things together

Like we mentioned before the binary load-time and runtime conventions were developed individually by Christopher and myself. To achieve binary compatibility these two conventions need to be followed in together, meaning our work must be combined.

7.6.1 Summary of work for Load-time Binary Compatibility

The hypervisor, Uhyve, was modified to load the kernel and the application as required to emulate the address space of Linux running an application. The position at which the kernel was changed, this position is defined by a hardcoded value. The offset of the loadable sections of the ELF formatted binary of the application is read from this same file. It is then used to load in these sections within the virtual memory address space according to these offsets. At this point several values are then passed to the kernel during its initialisation. These values include the entry point of the application and the path to the binary executable of the application on the filesystem of the host OS. An auxiliary vector is created within the stack and is initialised with values required by the application, these values include OS specific information and so these are hardcoded to mimic those of Linux. And finally, a jump is made to the entry point of the application, beginning the execution of the application.

Several bugs were encountered when initialising the stack and because of those the information on the stack were corrupted. These values are required by the C runtime which is run before the application is. Because the stack values are corrupted, the C runtime cannot be executed properly and fails to set up the environment for the execution for application, effectively making it not possible to run applications requiring the values on the stack. This means we are now in a state where we can only run assembly applications which do not use a C library.

7.6.2 Final State of Project

Due to the deadline marking the end of the project development approaching, we could not migrate all of our work done to Hermit-playground. This meant that we have provided partial binary compatibility to RustyHermit and slightly extended the number of system calls on Hermit-playground to include fstat, readv, writev, mmap, mprotect, munmap, brk, ioctl uname and arch_prctl.

7.7 Results

This section demonstrates the results of what was achieved.

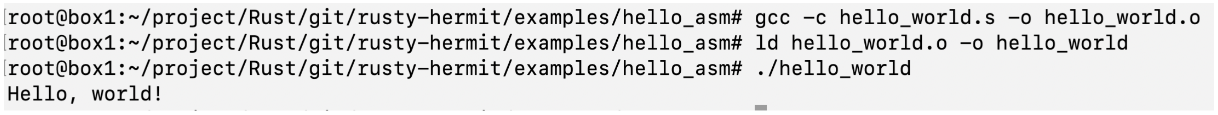
7.7.1 Binary Compatible RustyHermit

In the case of RustyHermit, we were able to achieve binary compatibility when it comes to running binaries of assembly applications for Linux, more specifically assembly applications which do not make use of a C library. Several assembly applications were developed to demonstrate this.

Hello World

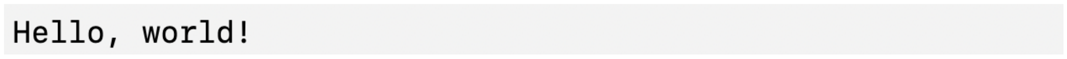
The first of these is unsurprisingly a "Hello world" application and the code is demonstrated in figure (7.4). After compiling the application written in a file called "hello_world.s" with commands `gcc -c hello_world.s -o hello_world.o` and `ld hello_world.o -o hello_world` we can run the executable on Linux with `./hello_world`. Figure (7.13) shows the output of running the binary on Linux and will thus be used to compare with the results from running this binary on top of our binary compatible unikernel.

Figure (7.14) shows the last line of output of the application running on top of RustyHermit. The other lines not included in the figure shows information from the kernel related to the loadable segments of the binary when the kernel is initialised, this can be ignored. We can see that the output of the application and it is identical to that of running it on Linux.



```
root@box1:~/project/Rust/git/rusty-hermit/examples/hello_asm# gcc -c hello_world.s -o hello_world.o
root@box1:~/project/Rust/git/rusty-hermit/examples/hello_asm# ld hello_world.o -o hello_world
root@box1:~/project/Rust/git/rusty-hermit/examples/hello_asm# ./hello_world
Hello, world!
```

Figure 7.13: Compiling and running Hello_world application in assembly on Linux.

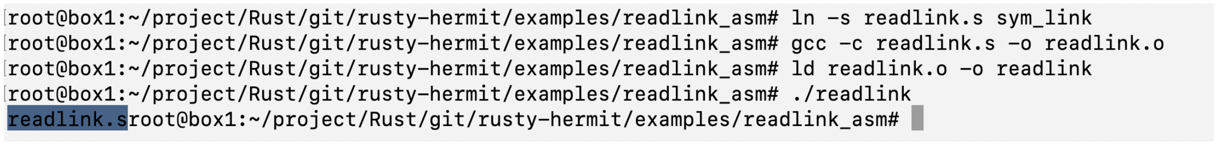


```
Hello, world!
```

Figure 7.14: Output of binary compatible RustyHermit from running Hello_world application in assembly, removing kernel initialisation messages.

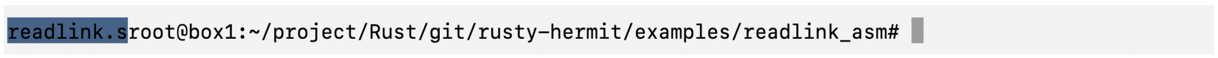
Readlink

This application outputs the target of a symbolic link and in saving space, its code is shown in the appendix at A.0.2. Following along from the figure (7.15), we can see that a symbolic link "sym_link" is created and its target is identified as the application's source "readlink.c". We then compile and run the application on Linux like before and the output is also shown in figure (7.16). Not that the output is highlighted to make it distinguishable from the command prompt.



```
root@box1:~/project/Rust/git/rusty-hermit/examples/readlink_asm# ln -s readlink.s sym_link
root@box1:~/project/Rust/git/rusty-hermit/examples/readlink_asm# gcc -c readlink.s -o readlink.o
root@box1:~/project/Rust/git/rusty-hermit/examples/readlink_asm# ld readlink.o -o readlink
root@box1:~/project/Rust/git/rusty-hermit/examples/readlink_asm# ./readlink
readlink.sroot@box1:~/project/Rust/git/rusty-hermit/examples/readlink_asm#
```

Figure 7.15: Compiling and running assembly application which uses readlink on Linux



```
readlink.sroot@box1:~/project/Rust/git/rusty-hermit/examples/readlink_asm#
```

Figure 7.16: Output of binary compatible RustyHermit running assembly application which uses readlink, removing kernel initialisation messages.

Now running this on as a unikernel. Figure (7.14) shows the output whilst removing the output from the kernel initialisation of the kernel mentioned before. The result is again identical to that of Linux.

More applications

A general theme has now been built when it comes to comparing the outputs of the applications on Linux and as a unikernel, the outputs are identical when we leave out the kernel initialisation messages. Although more applications have been developed to further demonstrate the binary capability of the new RustyHermit, we shall not demonstrate them but however leave their implementations in the appendix.

Chapter 8

Evaluation

8.1 What was achieved?

The main objective for the project was to extend RustyHermit to be binary compatible with Linux binaries, whilst taking advantage of the memory safe features of a memory safe programming language, Rust. At the end of the project we were able to achieve a unikernel which was able to run a set binaries of Linux applications written in assembly. We were able to achieve binary compatibility to a subset of applications we wanted to support, namely “Hello world” in assembly, show the Results sections. Along the way we chose to also begin to migrate our work to another unikernel, Hermit-playground. Most of the detailed objectives when it comes to binary compatibility were achieved and only one of the original applications we wanted to support were actually supported, namely “Hello world” in assembly.

8.2 What was not achieved?

We were unable to support all of our objectives in providing binary compatibility, due to issues in crafting the stack and migrating our code between different unikernels. So we were not able to support the full set of applications we meant to, and even if those issues did not arise, we would still need more system calls to be developed to support those applications.

8.3 Aims

With all this work done and the results shown under the Results section, we show that binary compatibility is feasible for RustyHermit in Rust, even if we only scratch the surface of full binary compatibility.

8.4 Is this a good solution?

The final part of this chapter looks into evaluating the solution we proposed and developed.

8.4.1 Is a Memory Safe Programming Language Worth it?

Performing this extension on a Rust based unikernel allows us to explore the benefits of memory safe programming languages. However, at this stage of the project using Rust did not provide as many benefits as I would have expected. Even if I was actively trying minimise the number of unsafe code blocks being used in my code, I still found myself using them quite often, and this slowed down the development as it is another construct that needs to be added to the code. Unsafe code blocks bypass compile time checks, and so these memory safety guarantees are no longer held up within them. That being said, the problems I have mentioned may only be specific to the development of system calls, as they do require working with raw pointers in registers as mentioned previously, and these are inherently unsafe and require the use of unsafe code blocks. To answer the question, I do believe memory safety is worth it and would benefit the unikernel as a whole. However, when it comes to system call handling, interfacing with the application at a binary level at runtime will always prove to be unsafe due the need to manipulate raw pointers.

8.4.2 Does it solve the problems of why unikernels lack widespread use?

At this point of the project, providing only binary compatibility to assembly applications completely eliminates the porting effort of running the application as a unikernel. This is shown by the binaries running on top of RustyHermit without changing them. However, we knew the problem of porting the application was going to be mostly fixed by providing binary compatibility when we looked at Hermitux [16], this means the answer to the question comes down to if the problems of a unikernel written in a memory unsafe programming language is solved by using memory safe one. The answer to this is yes as a whole, as we saw in answering the previous question, even if memory safety cannot always be guaranteed when dealing with system calls.

Chapter 9

Conclusion & Future Works

9.1 Conclusion

In this project we demonstrate the feasibility in extending a unikernel written in a memory safe programming language to be binary compatible in an attempt to overcome the main problems behind lack of widespread use of unikernels. The unikernel was RustyHermit, written in the Rust programming language, and achieved the first steps of binary compatibility by extending it to be binary compatible to a set of binaries of assembly applications natively compiled for Linux.

9.2 Future Work

Moving forward the project could be further developed to support C applications, this can be done continuing from where we left off, if this is done the task of supporting new C applications only involves developing the system calls required by that application. Like we mentioned before, with LWIP the development of system calls performing network specific tasks will be easier and would allow the project to support networking.

The binaries of those C applications mentioned before will have to be statically linked, in the future the project can be extended to support dynamically linked binaries which would allow a greater range of applications to be run on the unikernel. This can also be done by fully implementing the mmap system call which would be used by the dynamic loader which forms part of the C runtime and maps the dynamically linked library code into the virtual memory address space of the unikernel.

A unikernel friendly filesystem can be implemented within the address space of the virtual machine, this can be used to load in the files from the host required by the applications in runtime in the hopes of reducing the dependency on the host OS. This would have the effect of increasing the security of the unikernel as we interact with the host less often.

The potential of the project ranges from increasing the range of applications supported, to increasing the OS specific elements it incorporates in an attempt to reduce dependence on the host OS and so developing it into a full fledged operating system while still maintaining the unikernel principles.

Bibliography

[1] Harek Haugerud Paal E. Engelstad Alfred Bratterud, Alf-Andre Walla and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel for cloud services.

[2] Glauber Costa Pekka Enberg Nadav Har'El Don Marti Avi Kivity, Dor Laor, Vlad Zolotarov, and Cloudius Systems. Osv—optimizing the operating system for virtual machines.

[3] Florian Schmidt Jose Mendes Simon Kuenzer Sumit Sati Kenichi Yasukata Costin Raiciu Filipe Manco, Costin Lupu and Felipe Huici. My vm is lighter (and safer) than your container.

[4] Linux Foundation. Elf headers. https://refspecs.linuxfoundation.org/elf/gabi4+/ch4.eheader.html#elfid (accessed: 25.04.2021).

[5] Linux Foundation. Tool interface standard (tis) executable and linking format (elf) specification, version 1.2.

[6] Github. Hermit-playground. https://github.com/hermitcore/hermit-playground (accessed: 29.04.2021).

[7] Intel. Intel® 64 and ia-32 architectures software developer's manual, combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d and 4.

[8] Craig Sweetmore Samuel Rowe Hanan Hindy Christos Tachtatzis Robert Atkinson Joshua Talbot, Przemek Pikula and Xavier Bellekens. A security perspective on unikernels.

[9] Ryan Levick and Sebastian Fernandez. We need a safer systems programming language. https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/ (accessed: 25.04.2021).

[10] Anil Madhavapeddy and David J. Scott. Unikernels: The rise of the virtual library operating system.

[11] Andreas Jaeger Michael Matz1, Jan Hubicˇka, Don Marti Mark Mitchell, Vlad Zolotarov, and Cloudius Systems. System v application binary interface, draft version 0.99.7.

[12] Daniel Munoz. After all these years, the world is still powered by c programming. https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming (accessed: 24.04.2021).

[13] Hacker News. Java garbage collection can be really slow. https://news.ycombinator.com/item?id=11554700 (accessed: 25.04.2021).

[14] Hacker News. Unikernels are secure. https://news.ycombinator.com/item?id=14736909 (accessed: 10.04.2021).

[15] Pierre Olivier. Uname. https://github.com/ssrg-vt/hermitux-kernel/blob/ce593557e7ea046b15adb8be53f12120cc203ca2/kernel/syscalls/uname.c (accessed: 29.04.2021).

[16] Stefan Lankes Changwoo Min Pierre Olivier, Daniel Chiba and Binoy Ravindran. A binary-compatible unikernel.

[17] Ali Raza. Ukl: A unikernel based on linux. https://next.redhat.com/2018/11/14/ukl-a-unikernel-based-on-linux/ (accessed: 9.04.2021).

[18] Filipe Manco Jose Mendes Yuri Volchkov Florian Schmidt Kenichi Yasukata Michio Honda Simon Kuenzer, Anton Ivanov and Felipe Huici. Unikernels everywhere: The case for elastic cdns.

[19] Jens Breitbart Stefan Lankes and Simon Pickartz. Exploring rust for unikernel development.

[20] Simon Pickartz Stefan Lankes and Jens Breitbart. Hermitcore: a unikernel for extreme scale computing.

[21] Aaron Yi Ding Vittorio Cozzolino and Jörg Ott. Fades: Fine-grained edge offloading with unikernels.

[22] Xen Website. Unikraft. https://xenproject.org/developers/teams/unikraft/ (accessed: 29.04.2021).

[23] Wikipedia. Lwip. https://en.wikipedia.org/wiki/LwIP (accessed: 29.04.2021).

[24] Wikipedia. Memory saftey. https://en.wikipedia.org/wiki/Memory_safety (accessed: 28.04.2021).

[25] Wikipedia. Micropython. https://en.wikipedia.org/wiki/MicroPython (accessed: 29.04.2021).

[26] Wikipedia. Musl. https://en.wikipedia.org/wiki/Musl (accessed: 29.04.2021).

[27] Wikipedia. Nas parallel benchmarks. https://en.wikipedia.org/wiki/NAS_Parallel_Benchmarks (accessed: 29.04.2021).

[28] Wikipedia. Rust (programming language). https://en.wikipedia.org/wiki/Rust_(programming_language) (accessed: 25.04.2021).

Appendix A

Code for Assembly Applications Supported

This appendix shows the code of the assembly applications which our binary compatible Rusty-Hermit is able run.

A.0.1 Hello World

The following listing shows the assembly code for ”hello world” application support by our unikernel.

```

        .global _start

        .text
_start:
      # write(STDOUT, message, 14)    now we print the value to standard
    output
        mov     SYS_WRITE, %rax
        mov     STDOUT, %rdi
        mov     $message, %rsi
        mov     $14, %rdx
        syscall                          # invoke operating system to do the
     write

exit:   # exit(0)                  # terminate the program
        mov     SYS_EXIT, %rax
        xor     %rdi, %rdi               # we want return code 0
        syscall

.section  .data
message:
        .ascii  "Hello, world!\n"
.skip 8

STDOUT: .word   1
.skip 8
SYS_WRITE:      .word   1
.skip 8
SYS_EXIT:       .word   60
.skip 8
```

A.0.2 Readlink

The following listing shows the assembly application which our unikernel supports. This invokes, readlink, write and exit system calls.

```
        .global _start

        .text
_start:

  # readlink(fd, buf, 12) # perform readlink on the symbolic link
        mov     SYS_READLINK, %rax
        mov     $link_path, %rdi
        mov     $buf, %rsi
        mov     buf_len, %rdx
        syscall       # invoke operating system to do the readlink

  cmp $0, %rax    # if 0 bytes were read from the symbolic link we have failed,
  je  exit      # close the file and terminate the program

  add $1, %rax    # add a new line
  mov %rax, readlink_len

  # write(STDOUT, buf, readlink_len)  # if we did not fail, print to standard output what readlink read
        mov     SYS_WRITE, %rax
        mov     STDOUT, %rdi
        mov     $buf, %rsi
        mov     readlink_len, %rdx
        syscall                         # invoke operating system to do the write

exit:   # exit(0)     # terminate the program
        mov     SYS_EXIT, %rax
        xor     %rdi, %rdi              # we want return code 0
        syscall

.section        .data
link_path:
        .ascii  "sym_link\0"
.skip 8
buf_len:
  .word 20
.skip 8

STDOUT: .word   1
.skip 8
SYS_READLINK:   .word   89
.skip 8
SYS_WRITE:      .word   1
.skip 8
SYS_EXIT:       .word   60
.skip 8

.section  .bss
.lcomm len, 8
```

```
.lcomm buf, 20
.lcomm readlink_len, 8
```

A.0.3 File Management System Calls

This assembly application is supported by our unikernel and invokes the following system calls: open, read, write, lseek close and exit. All of these can be considered file management system calls except for exit.

```
        .global _start

        .text
_start:
        # open(filename, flags, mode) # open a file
        mov     SYS_OPEN, %rax
        mov     $filename, %rdi
        mov     flags, %rsi
        mov     mode, %rdx
        syscall                         # invoke operating system to do the open

        cmp     $0, %rax                # if file descriptor is less than or equal to zero
        jle     exit                    # jump to exit and terminate
        mov     %rax, fd


  # read(fd, inputString, 12) # read from the file  into inputString
        mov     SYS_READ, %rax
        mov     fd, %rdi
        mov     $inputString, %rsi
        mov     $12, %rdx
        syscall       # invoke operating system to do the read

  cmp $12, %rax   # if 12 bytes were not read something went wrong,
  jne close     # close the file and terminate the program


  # write(fd, inputString, 12)  # now we print the value to standard output
        mov     SYS_WRITE, %rax
        mov     STDOUT, %rdi
        mov     $inputString, %rsi
        mov     $12, %rdx
        syscall                         # invoke operating system to do the write

  cmp     $12, %rax
        jne     close


  # lseek(fd, newOffset, whence)  # move the file offset to the beginning of the file
  mov     SYS_LSEEK, %rax
        mov     fd, %rdi
        mov     $-5, %rsi
        mov     $1, %rdx    # We want to set the file offset the current
   offset + newOffset
        syscall

  cmp   $0, %rax
  jl  close

  # write(fd, message, 7)        # write message to the file at the new
   file offset
        mov     SYS_WRITE, %rax
        mov     fd, %rdi
        mov     $message, %rsi
        mov     $7, %rdx
        syscall

close:  # close(fd)     # close the file
        mov     SYS_CLOSE, %rax
        mov     fd, %rdi
  syscall

exit:   # exit(0)     # terminate the program
        mov     SYS_EXIT, %rax
        xor     %rdi, %rdi              # we want return code 0
        syscall

.section        .data
message:
        .ascii  "world!\n"
.skip 8
filename:
        .ascii  "file.txt\0"

.skip 8
flags:  .word 0x0102
.skip 8
mode:   .word 0x309
.skip 8

STDOUT: .word   1
.skip 8
SYS_READ:       .word   0
.skip 8
SYS_WRITE:      .word   1
.skip 8
SYS_OPEN:       .word   2
.skip 8
SYS_CLOSE:      .word   3
.skip 8
SYS_LSEEK:       .word   8
.skip 8
SYS_EXIT:       .word   60
.skip 8

.section  .bss
.lcomm fd, 8
.lcomm inputString, 12
```

A.0.4 Memory Management System Calls

This assembly application is supported by our unikernel and invokes the following system calls: mmap, write, munmap. Mmap and munmap are two memory management system calls.

```
        .global _start

        .text
_start:
        # mmap(addr, len, prot, flags, fd, offset)  #  Request memory for read and write of length len
        mov     SYS_MMAP, %rax
        mov     $0, %rdi
        mov     len, %rsi   # amount of memory we want
        mov     prot, %rdx    # read & write
  mov flags, %r10   # private & map_anonymous
  mov $-1, %r8    # fd (not needed)
  mov $0, %r9     # offset (not needed)
        syscall                         # invoke operating system to do the mmap

        cmp     $0, %rax                # if the address of the block of memory allocated is zero then it failed
        je     exit                     # jump to exit and terminate
        mov     %rax, addr



  # write(STDOUT, message, 9)    # now we print the value to standard output
        mov     SYS_WRITE, %rax
        mov     STDOUT, %rdi
        mov     $message, %rsi
        mov     $9, %rdx
        syscall                         # invoke operating system to do the write



  # munmap(addr, len)      #  Request memory allocated to be freed
        mov     SYS_MUNMAP, %rax
        mov     addr, %rdi    # address of the block of memory that was allocated
        mov     len, %rsi
  syscall

        cmp     $0, %rax               # if the address of the block of memory allocated is zero then it failed
        je     exit                    # jump to exit and terminate
        mov     %rax, addr


exit:   # exit(0)     # terminate the program
        mov     SYS_EXIT, %rax
        xor     %rdi, %rdi              # we want return code 0
        syscall

```

```
.section        .data
message:
        .ascii  "Success!\n"
.skip 8
len:  .word 1024
.skip 8
prot: .word 0x3
.skip 8
flags:  .word 0x022   # private & map_anonymous
.skip 8
mode:   .word 0x309
.skip 8

STDOUT: .word   1
.skip 8
SYS_READ:       .word   0
.skip 8
SYS_WRITE:      .word   1
.skip 8
SYS_MMAP:       .word   9
.skip 8
SYS_MUNMAP:      .word   11
.skip 8
SYS_MPROTECT:    .word   10
.skip 8
SYS_EXIT:       .word   60
.skip 8

.section  .bss
.lcomm addr, 8
.lcomm inputString, 12
```